Objects and Classes

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

August 26, 2024

- Seen two ways to initialise objects:
  1. A no-argument constructor can initialise data members to constant values, and
  2. a multi-argument constructor can initialise data members to values passed as arguments.
- Can we initialise it with another object of the same type?
- see `ecopycon.cpp`
- We initialise `dist1` to the value of 11’ -6.25” using the two-argument constructor.
- Then we define two more objects of type Distance, `dist2` and `dist3`,
- initialising both to the value of `dist1`.

The Default Copy Constructor

- this should require us to define a one-argument constructor ??
- but initializing an object with another object of the same type is a special case.
- We don't need to create a special constructor for this; one is already built into all classes.
- It's called the default copy constructor.
- It's a one-argument constructor whose argument is an object of the same class as the constructor.
- These definitions both use the default copy constructor.
- This causes the default copy constructor for the Distance class to perform a member-by-member copy of `dist1` into `dist2`. (for `dist3` as well)
- Although this looks like an assignment statement, it is not.
- Both formats invoke the default copy constructor, and can be used interchangeably.

```
Distance dist1(11, 6.25);   //two-arg constructor
Distance dist2(dist1);  //one-arg constructor
Distance dist3 = dist1; //also one-arg constructor
```

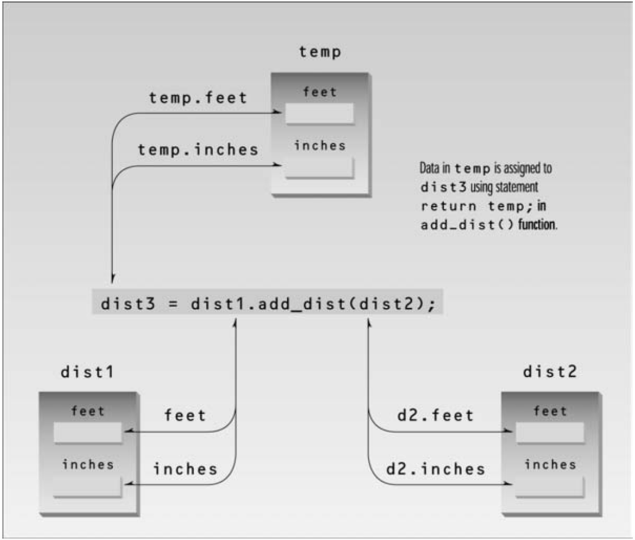
Returning Objects from Functions

- two distances were passed to `add_dist()` as arguments (see `// englcon.cpp`)
- and the result was stored in the object of which `add_dist()` was a member, namely `dist3`.
- see `englret.cpp`
- In `main()`, the result is assigned to `dist3` in the statement: `dist3 = dist1.add_dist(dist2);`

```
//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2)
{
   Distance temp;                       //temporary variable
   temp.inches = inches + d2.inches;    //add the inches
   if(temp.inches >= 12.0)              //if total exceeds 12.0,
   {                                    //then decrease inches
      temp.inches -= 12.0;              //by 12.0 and
      temp.feet = 1;                    //increase feet
   }                                    //by 1
   temp.feet += feet + d2.feet;         //add the feet
   return temp;
}
```

- a temporary object of class Distance is created.
- This object holds the sum until it can be returned to the calling program.
- The sum is calculated by adding two distances.
- The first is the object of which `add_dist()` is a member, `dist1`.
- Its member data is accessed in the function as feet and inches.
- The second is the object passed as an argument, `dist2`.
- Its member data is accessed as `d2.feet` and `d2.inches`.
- The result is stored in temp and accessed as `temp.feet` and `temp.inches`.
- The temp object is then returned by the function using the statement
- Notice that `dist1` is not modified; it simply supplies data to `add_dist()`.

Result returned from the temporary object



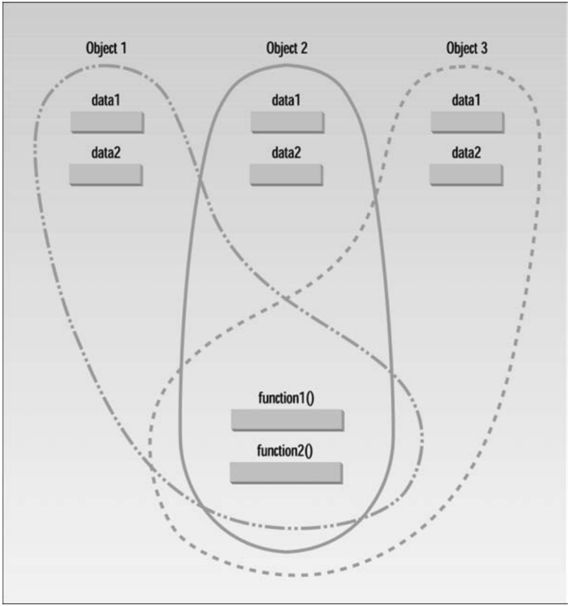
- see `cardobj.cpp`, it does not introduce any new concepts
- but it does use almost all the programming ideas we've discussed up to this point.
- `CARDOBJ` creates three cards with fixed values and switches them around in an attempt to confuse the user about their location.
- in `CARDOBJ` each card is an object of class card.
- The `isEqual()` function checks whether the card is equal to a card supplied as an argument.
- It uses the conditional operator to compare the card of which it is a member with a card supplied as an argument.
- This function could also have been written with an `if...else statement`
- but the conditional operator is more compact.

```
if( number == c2.number && suit == c2.suit )
  return true;
else
  return false;
```

- There is an impression that each object created from a class contains separate copies of that class's data and member functions.
- This is a good first approximation, since it emphasises that objects are complete, self-contained entities, designed using the class definition.
- The mental image here is of `cars` (objects) rolling off an assembly line, each one made according to a `blueprint` (the class definitions).
- Actually, things are not quite so simple.
- It's true that each object has its own separate data items.
- On the other hand, contrary to what you may have been led to believe, all the objects in a given class use the same member functions.
- The member functions are created and placed in memory only once when they are defined in the class definition.

- This makes sense;
- there's really no point in duplicating all the member functions in a class every time you create another object of that class, since the functions for each object are identical.
- The data items, however, will hold different values, so there must be a separate instance of each data item for each object.
- Data is therefore placed in memory when each object is defined, so there is a separate set of data for each object.
- In the `SMALLOBJ` example, there are two objects of type `smallobj`, so there are two instances of `somedata` in memory.
- However, there is only one instance of the functions `setdata()` and `showdata()`
- These functions are shared by all the objects of the class.

Objects, data, functions, and memory



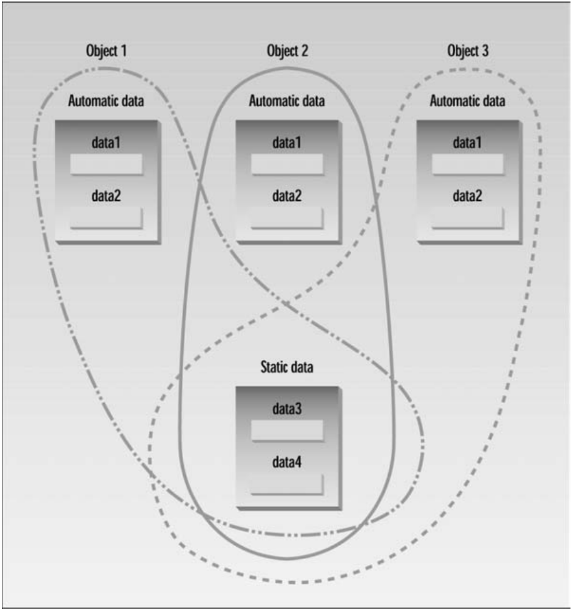
Static Class Data

- If a data item in a class is declared as static, only one such item is created for the entire class, no matter how many objects there are. (see `statdata.cpp`)
- A static data item is useful when all objects of the same class must share a common item of information.
- A member variable defined as `static` has characteristics similar to a normal `static` variable:
- It is visible only within the class, but its lifetime is the entire program.
- It continues to exist even if there are no objects of the class.
- static class member data is used to share information among the objects of a class.
- As an example, suppose an object needed to know how many other objects of its class were in the program.
- In a road-racing game, for example, a race car might want to know how many other cars are still in the race.
- In this case a static variable count could be included as a member of the class.

Static Class Data

- The class foo in this example has one data item, count, which is type `static int`.
- constructor for this class causes count to be incremented.
- In `main()` we define three objects of class foo.
- Since the constructor is called three times, count is incremented three times.
- the member function, `getcount()`, returns the value in count.
- We call this function from all three objects, and—as we expected—each prints the same value
  `count is 3` $\longrightarrow$ static data
- If we had used an ordinary automatic variable—as opposed to a static variable—for count, each constructor would have incremented its own private copy of count once, and the output would have been
  `count is 1` $\longrightarrow$ automatic data

Static versus automatic member variables.



Separate Declaration and Definition

- Static member data requires an unusual format.
- Ordinary variables are usually declared (the compiler is told about their name and type) and defined (the compiler sets aside memory to hold the variable) in the same statement.
- Static member data, on the other hand, requires two separate statements.
- The variable's declaration appears in the class definition, but the variable is actually defined outside the class, in much the same way as a global variable.
- Why is this two-part approach used?
- If static member data were defined inside the class, it would violate the idea that a class definition is only a blueprint and does not set aside any memory.
- Putting the definition of static member data outside the class also serves to emphasise that the memory space for such data is allocated only once, before the program starts to execute,
- and that one static member variable is accessed by an entire class;
- In this way a static member variable is more like a global variable.

- `const` used on normal variables to prevent them from being modified,
- and `const` can be used with function arguments to keep a function from modifying a variable passed to it by reference.
- We can introduce some other uses of `const`: on member functions, on member function arguments, and on objects.
- A `const` member function guarantees that it will never modify any of its class's member data.

```
//constfu.cpp
//demonstrates const member functionsxw
class aClass
{
private:
    int alpha;
public:
    void nonFunc()        //non–const member function
        { alpha = 99; }   //OK
    void conFunc() const  //const member function
        { alpha = 99; }   // ERROR: can't modify a member
};
```

- The `non-const` function `nonFunc()` can modify member data alpha, but the constant function `conFunc()` can't.
- If it tries to, a compiler error results.
- A function is made into a constant function by placing the keyword `const` after the declarator but before the function body.
- If there is a separate function declaration, `const` must be used in both declaration and definition.
- Member functions that do nothing but acquire data from an object are obvious candidates for being made `const`, because they don't need to modify any data.
- Making a function `const` helps the compiler flag errors, and tells anyone looking at the listing that you intended the function not to modify anything in its object.
- see `engConst.cpp`

`const` Member Function Arguments

- if an argument is passed to an ordinary function by reference, and you don't want the function to modify it, the argument should be made const in the function declaration (and definition).
- This is true of member functions as well.
- the argument to `add_dist()` is passed by reference, and we want to make sure that it won't modified.
- Therefore we make the argument d2 to `add_dist()` const in both declaration and definition.

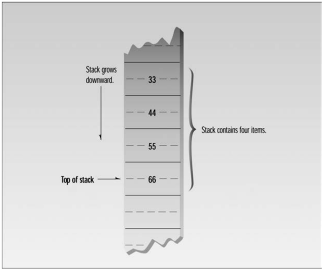
- We've seen that we can apply `const` to variables of basic types such as `int` to keep them from being modified.
- In a similar way, we can apply `const` to objects of classes.
- When an object is declared as `const`, we can't modify it.
- We can use only `const` member functions with it, because they're the only ones that guarantee not to modify it.
- see `constObj.cpp`
- The `CONSTOBJ` program makes football a `const` variable.
- Now only `const` functions, such as `showdist()`, can be called for this object.
- Non-const functions, such as `getdist()`, which gives the object a new value obtained from the user, are illegal.
- While designing classes it's a good idea to make `const` any function that does not modify any of the data in its object.
- This allows the user of the class to create `const` objects.
- These objects can use any `const` function, but cannot use any `non-const` function.

Arrays as Class Member Data

- Arrays can be used as data items in classes.
- Let's look at an example that models a common computer data structure: the stack.
- *A stack works like the spring-loaded devices that hold trays in cafeterias. When you put a tray on top, the stack sinks down a little; when you take a tray off, it pops up. The last tray placed on the stack is always the first tray removed.*
- see `stakaray.cpp`
- The important member of the stack is the array `st`.
- An `int` variable, top, indicates the index of the last item placed on the stack; the location of this item is the top of the stack.
- The size of the array used for the stack is specified by `MAX`, in the statement
- it's preferable to define constants that will be used entirely within a class, as `MAX` is here, within the class.

Arrays as Class Member Data

- Thus the use of global const variables for this purpose is nonoptimal.
- Standard C++ mandates that we should be able to declare `MAX` within the class as:
  `static const int MAX = 10;`
- This means that `MAX` is constant and applies to all objects in the class.



- Since memory grows downward in the figure, the top of the stack is at the bottom in the figure.

- When an item is added to the stack, the index in top is incremented to point to the new top of the stack.
- When an item is removed, the index in top is decremented.
- We don't need to erase the old value left in memory when an item is removed; it just becomes irrelevant.
- To place an item on the stack - call the `push()`
- To retrieve - use the `pop()`

- We can create an array of objects.
- see `englaray.cpp`
- In this program the user types in as many distances as desired.
- After each distance is entered, the program asks if the user desires to enter another.
- If not, it terminates, and displays all the distances entered so far.
- A class member function that is an array element is accessed by the dot operator:
  `dist[j].showdist();`
- The array name followed by the index in brackets is joined, using the dot opera- tor, to the member function name followed by parentheses.